Parallel Implementation of Constraint Solving*

Alvaro Ruiz-Andino[1], Lourdes Araujo[1], Fernando Sáenz[2], and José Ruz[2]

[1] Department of Computer Science
University Complutense of Madrid, Spain
{alvaro,lurdes}@sip.ucm.es
[2] Department of Computer Architecture
University Complutense of Madrid, Spain
{fernan,jjruz}@eucmax.sim.ucm.es

Abstract. Many problems from artificial intelligence can be described as constraint satisfaction problems over finite domains (CSP(FD)), that is, a solution is an assignment of a value to each problem variable such that a set of constraints is satisfied. Arc-consistency algorithms remove inconsistent values from the set of values that can be assigned to a variable (its domain), thus reducing the search space. We have developed a parallelisation scheme of arc-consistency to be run on MIMD multiprocessor. The set of constraints is divided into N partitions, which are executed in parallel on N processors. The parallelisation scheme has been implemented on a CRAY T3E multiprocessor with up to thirty-four processors. Empirical results on speedup and behaviour are reported and discussed.

1 Introduction

Constraint Programming over finite domains (CP(FD)) has been used for specifying and solving complex constraint satisfaction and optimisation problems, such as resource allocation, scheduling and hardware design [8]. Finite domain Constraint Satisfaction Problems (CSP) usually describe NP-complete search problems, but it has been shown that by working locally on constraints and their related variables it is possible to dynamically prune the search space in an efficient way. Techniques following this approach, called arc-consistency (AC) algorithms, eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space both before and while searching. Waltz [9] proposed the first arc-consistency algorithm, and several improved versions are described in the literature: AC-5 [7], and AC-6 [1].

We have developed and tested a parallelisation scheme of arc-consistency for MIMD distributed shared memory multiprocessors. The set of constraints is partitioned into N sets, which are processed in parallel on N processors.

Several parallel processing methods for solving CSP's have been proposed. In [11], a parallel constraint solving technique for a special class of CSP, acyclic

* Supported by project CICYT-TIC98-0445-C03-02/97

V. Malyshkin (Ed.): PaCT-99, LNCS 1662, pp. 466–472, 1999.

constraint networks, is developed. It also presents some results on parallel complexity, generalising results in [3]. In [4], it is concluded that parallel complexity of constraint networks is critically dependent on subtle properties of the network which do not influence its sequential complexity. They propose massively parallel processing of arc-consistency with also very simple processing elements.

In [2,5] Nguyen, Deville and Baudot proposed distributed versions for AC-3, AC-4, and AC-6 for binary CSP's. Instead, our work is focused on AC-5, and, we report empirical data obtained running the parallel arc-consistency algorithms on a CRAY T3E, a distributed shared memory multiprocessor.

1.1 Constraint Programming

A constraint satisfaction problem over finite domains may be stated as follows. Given a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where

- $\mathcal{V} \equiv \{v_1, \cdots, v_n\}$, is a set of domain variables,
- $\mathcal{D} \equiv \{d_1, \cdots, d_n\}$, is the set of an initial *finite domain* (finite set of values) for each variable,
- $\mathcal{C} \equiv \{c_1, \cdots, c_m\}$, is a set of constraints among the variables in $\mathcal{V}$. A constraint $c \equiv (V_c, R_c)$ is defined by a subset of variables $V_c \subseteq \mathcal{V}$, and a subset of allowed tuples of values $R_c \subseteq \bigotimes_{v_i \in V_c} d_i$, where $\bigotimes$ denotes Cartesian product.

The goal is to find an assignment for each variable $v_i \in \mathcal{V}$ of a value from each $d_i \in \mathcal{D}$ which satisfies every constraint $c_i \in \mathcal{C}$. Besides the explicit relational constraints, the allowed constraints usually include arithmetic ones as well as some specific symbolic constraints used in classic resource allocation problems like scheduling and packing.

A constraint $c \equiv (V_c, R_c) \in \mathcal{C}$, $V_c \equiv \{v_1, \cdots, v_k\}$, is *arc-consistent* with respect to domains $\{d_1, \cdots, d_k\}$ iff for all $v_i \in V_c$, for all $a \in d_i$, there exists a tuple $(b_1, \cdots, b_{i-1}, a, b_{i+1}, \cdots, b_k) \in R_c$, where $b_j \in d_j$. A CSP is called arc-consistent iff all $c_i \in \mathcal{C}$ are arc-consistent with respect to $\mathcal{D}$.

The starting point of this work is a sequential constraint solver which implements the AC-5 arc-consistency algorithm [7]. AC-5 *revises* constraints removing inconsistent values from the domains of the variables until either a fixed point is reached, or inconsistency is detected. A propagation queue is used to schedule the revision of constraints. As the result of revising a constraint the domain of a variable may be pruned, and in such a case the variable is queued. Termination, correctness, complexity, and properties of the algorithm have been studied extensively in the literature [7]. Correctness is independent of the order of revising the constraints, which constitutes the basis for the correctness of the parallel version of the algorithm.

The rest of the paper is organised as follows. Next section describes the parallel execution scheme. Section 3 reports and discusses the experimental results. Finally, conclusions are drawn in section 4.

2 Parallel Arc-Consistency

The arc-consistency algorithm presents an inherent parallelism. Each constraint behaves as a concurrent process which updates the domains of variables, triggered by changes in the domains of other variables. There is an inherent sequentiality, as well, since a constraint must be revised only as the consequence of a previous revision of another constraint. This sequentiality defines a partial order among revising constraints. A constraint is *ready* if any of its variables has been pruned after its last revision. At any time during the execution of the arc-consistency algorithm there will be a set of ready constraints, called the *ready set*. In a sequential version of a consistency algorithm the ready set is stored in a *propagation queue* (updated whenever a variable is modified), assuring a sound execution order of constraints, that is, that a constraint is revised after the pruned variable has been updated. Parallel consistency algorithms simultaneously revise the constraints in the ready set, providing mechanisms to maintain a sound order.

A static partition ensures a sound order of revising constraints, since the parallel algorithm is basically the sequential one, but applied to a subset of the constraints. The only coordination mechanism needed by this scheme comes from the detection of termination, which can be carried out by one of the processors, called the *distinguished* one. The mapping of constraints to processors is generated previously to the execution of arc-consistency. An important factor for the efficiency of this scheme is the criterion for the distribution of constraints among processors, therefore different criteria have been investigated.

Parallelisation of the consistency algorithm requires every processor to have access to a common store for the domains of the variables. Since the presented parallelisation scheme is focused on distributed memory architecture, each processor will maintain a (partial) local copy of the store. Changes in the variables' domains must be communicated to concerned processors in order to maintain coherency among local copies of the domains.

The set of constraints $\mathcal{C}$ is partitioned into n disjoint subsets, $\mathcal{C} = C_1 \cup \cdots \cup C_n$. This partitioning induces a distribution of the set of domain variables $\mathcal{V}$ in n not necessarily disjoint subsets $V_1, \cdots, V_n$ ($\mathcal{V} = V_1 \cup \cdots \cup V_n$). For all constraints $c_j \in C_i$, the variables involved in c_j constitute V_i. Partitions $\langle V_i, D_i, C_i \rangle$ are mapped one-to-one to processing elements P_i. Each processing element P_i performs sequential arc-consistency, revising constraints belonging to C_i, and consequently updating local copies of variables in V_i. Since the distribution of the set of variables $\mathcal{V}$ is non-disjoint, some variables will be located at several processing elements. Therefore, each processing element P_i must broadcast the prunings of the domain of variable v to every processing element P_j which has been assigned any of those constraints which involve variable v. Upon receiving the notification, processing elements P_j intersect their local copies of the domain with the incoming domain, probably triggering further propagation. Communication among processors is also needed in order to detect termination of the algorithm, either because of reaching the global fixed point, or because of inconsistency detection.

2.1 Parallel Algorithm

The parallel arc-consistency algorithm, as the sequential one, is a fixed point algorithm. Every processor executes a copy of it, maintaining a private propagation queue. The main steps of the algorithm are:

1. Initialize the local propagation queue, as the result of the revision of the local constraints.
2. Repeat the following steps until the global fixed point is reached or inconsistency is detected:
   - Revise local constraints until either the local propagation queue is empty (local fixed point) or inconsistency is detected.
   - Notify local fixed point to the *distinguished* processor, and wait until either:
     - Other processor communicates a change in the domain of a variable, therefore the local fixed point is left and revision of constraints continues.
     - The distinguished processor communicates that the global fixed point has been reached.
     - Other processor communicates inconsistency.

Whenever the revision of a local constraint results in the modification of the domain of a variable v, the processor broadcasts a message to the set of processors that have been assigned any of those constraints which involve variable v. Upon receiving the message these processors either detect inconsistency or properly update their local propagation queue and their local copy of variable v. Whenever a processor detects inconsistency, it broadcasts the failure to the rest of processors. Inconsistency is detected whenever:

- an empty domain results from the revision of a local constraint.
- an empty domain results from the intersection of the local domain of a variable with the domain received from another processor.

The global fixed point is reached when every processor is in a local fixed point and there are no pending messages. The distinguished processor is the only one responsible for the detection of termination. However, it performs local propagation as any other processor. In order to be able to detect the global fixed point, processors must notify to the distinguished one whenever they reach a local fixed point –along with the number of messages they have sent and received– and whenever they leave it due to an incoming message. When termination is detected, the distinguished processor notifies to the rest of processors.

A synchronisation among all processors is needed at the beginning of the algorithm, just after the initialisation of the communication status variables. Another synchronisation is needed if the algorithm finishes with inconsistency detection; otherwise, the global fixed point detection implies a synchronisation among processors.

3 Experimental Results

The presented parallel algorithms have been written in C, and developed and tested on a CRAY T3E multiprocessor with thirty-four 400-MHz DEC Alpha processors, 128 Mb of memory per processor, under UNICOS (UNIX) operating system. Notification of failure, global and local fixed point detection, activity status, and number of messages sent and received, have been implemented using the remote memory write feature of the CRAY T3E multiprocessor (routines from CRAY's shared memory library). Queues of messages are used for receiving domain updates. Messages are broadcasted to queues also using the fast remote memory write feature.

Reported results correspond to the time required to reach the first or all solutions, depending on the benchmark, performing a first fail sequential labelling. Therefore, reported speedup is lower than speedup achieved in a single call to the arc-consistency algorithm, since the search for a solution usually comprises a large number of calls to the arc-consistency algorithm, executed in parallel, interleaved with the selection and assignment of a value to a variable, executed sequentially.

We report the results for two representative benchmarks from the set used to evaluate the performance of the presented parallelisation scheme:

1. *N-Queens* problem consists in placing N queens in an N×N chess board in such a way that no queen attacks each other. The instance presented corresponds to N = 111, size which leads to a significant execution time.
2. *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP). Synthetic parametrizable constraint satisfaction problems allow studying the performance of an arc-consistency algorithm as some significant problem parameters vary. Instances of this problem are randomly generated given four parameters: number of variables (nv), the size of the initial domains (ds), density, and tightness. Figure 1 reports results obtained for an instance of this problem where $nv = 100$, $ds = 20$, $density = 0.75$, and $tightness = 0.85$.

Charts in figure 1 show, for each benchmark, the speedup vs. the number of processors. It can be observed that whereas *PBCSP* problems present a nearly linear speedup, the speedup for *Queens* benchmark stops increasing from a certain number of processors. The main factor for this different behaviour are that in *PBCSP* benchmark calls to the arc-consistency algorithm have a larger execution time, and revision of constraints has a larger granularity. Besides, *PBCSP* has a constraint graph with a more uniform topology, leading to a better workload balance. In order to study this factor we have measured the minimum and the maximum number of constraints executed per processor. The difference between minimum and maximum indicates workload balance quality. For *PBCSP* benchmarks, the minimum and maximum values do not differ significantly, indicating a high balanced workload, whereas this is not the case for *Queens* benchmark.

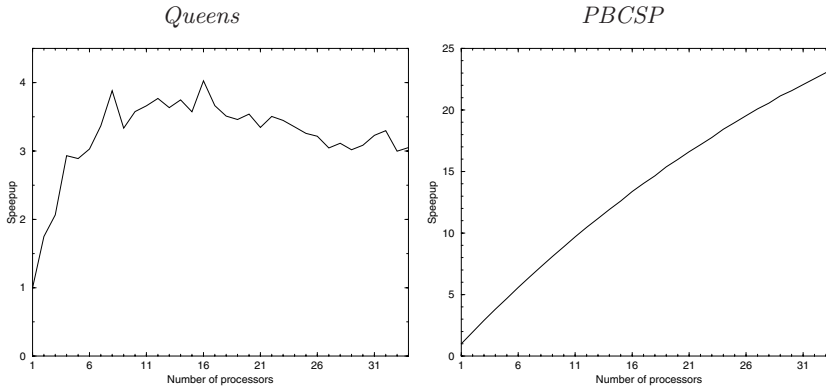

Fig. 1. Speedup curves for selected benchmarks.

4 Conclusions

We have developed and evaluated a parallelisation scheme of an arc-consistency algorithm for constraint satisfaction problems over finite domains. The scheme has been implemented on a CRAY T3E, a distributed shared memory MIMD multiprocessor, and empirical data are reported for two benchmarks.

The speedup obtained is nearly linear for *PBCSP* benchmarks, whereas for others speedup stops increasing from a problem dependent number of processors. This difference is mainly due to the more uniform constraint graph and larger granularity of *PBCSP* benchmarks, which leads to a better workload balance. In order to study how the performance of the parallel system depends on the characteristics of the constraint satisfaction problem to solve, the parametrizable synthetic benchmark has been tested for different sets of parameters. Results show that the system is better suited for large scale problems with a dense constraint graph.

References

1. Bessiere, D.: Arc-consistency and arc-consistency again. Artificial Intelligence Journal 65 (1994) 179-190. 466
2. Baudot, B., Deville, Y.: Analysis of Distributed Arc-Consistency Algorithms. Tech. Rep. 97-07. Uni. of Louvain, Belgium (1997). 467
3. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence 45 (1990) 275-286. 467
4. Kasif, S., Delcher, A. L.: Local Consistency in Parallel Constraint-Satisfaction Networks. Artificial Intelligence 69 (1994) 307-327. 467
5. Nguyen, T., Deville, Y.: A Distributed Arc-Consistency Algorithm. Science of Computer Programming, 30 (1998) 227-250. 467

6. Ruiz-Andino, A., Araujo, L., Ruz, J.: Parallel constraint satisfaction and optimisation. The PCSO system. Technical Report 71.98. Department of Computer Science. Universidad Complutense de Madrid (1998)
7. Van Hentenryck P., Deville, Y., Teng C. M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321. 466, 467
8. Wallace, M.: Constraints in Planning, Scheduling and Placement Problems. Constraint Programming, Springer-Verlag (1994). 466
9. Waltz, D.: Generating semantic descriptions for drawings of scenes with shadows. Technical Report AI271, MIT, Cambridge, MA. (1972). 466
10. Yokoo, M.: Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. Principles and Practice of Constraint Programming (1995) 88-102.
11. Zhang, Y., Mackworth, A. K.: Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Parallel Processing for Artificial Intelligence. Elsevier Science. (1993). 466